From Monolithic to Microservices An Experience Report from the Banking Domain

Bucchiarone, Antonio; Dragoni, Nicola; Dustdar, Schahram; Larsen, Stephan T.; Mazzara, Manuel

Published in:
IEEE Software

Link to article, DOI:
10.1109/MS.2018.2141026

Publication date:
2018

Document Version
Peer reviewed version

From Monolithic to Microservices: An experience report

Technical Report · August 2017

DOI: 10.13140/RG.2.2.34717.00482

5 authors, including:

Antonio Bucchiarone
Fondazione Bruno Kessler
104 PUBLICATIONS **1,165** CITATIONS
SEE PROFILE

Nicola Dragoni
Örebro universitet, Sweden, and Technical University of Denmark
119 PUBLICATIONS **1,396** CITATIONS
SEE PROFILE

Schahram Dustdar
TU Wien
25 PUBLICATIONS **1,381** CITATIONS
SEE PROFILE

Manuel Mazzara
Innopolis University
214 PUBLICATIONS **1,524** CITATIONS
SEE PROFILE

From Monolithic to Microservices: An experience report

Antonio Bucchiarone*, Nicola Dragoni†, Schahram Dustdar‡, Stephan T. Larsen§, Manuel Mazzara¶
* Fondazione Bruno Kessler, Trento, Italy
bucchiarone@fbk.eu
† Technical University of Denmark and Örebro University, Sweden
ndra@dtu.dk
‡ TU Wien
dustdar@dsg.tuwien.ac.at
§ Danske Bank, Denmark
stephantl@gmail.com
¶ Innopolis University, Russia
m.mazzara@innopolis.ru

***Abstract*—Microservices have seen their popularity blossoming with an explosion of concrete applications in real-life software. Several companies are currently involved in major refactoring of their back-end systems in order to improve scalability. In this paper, we present an experience report of a real world case study in order to demonstrate how scalability is positively affected by re-implementing a monolithic architecture into microservices. The case study is based on the *FX Core* system, a mission critical system of Danske Bank, the largest bank in Denmark and one of the leading financial institutions in Northern Europe.**

***Index Terms*—Microservices, Software Architecture, Scalability.**

I. Introduction

Microservices [10] is an architectural style originated from Service-Oriented Architectures (SOAs) [8] with the idea of bringing *in the small* (within an application) those concepts that worked *in the large*, i.e. for cross-organization business-to-business workflow. The shift towards microservices is a sensitive matter these days, seeing several companies involved in a major refactoring of their back-end systems to accommodate the advantages of the new paradigm. This is the case of the system and the institution considered in this paper, i.e., the *FX Core* of Danske Bank.

In monolithic architectures, the modularization abstractions rely on the sharing of resources of the same machine (memory, databases, files) and the components are therefore not independently executable. A notable problem of monoliths is *maintainability* and *evolvability*, and in general all the aspects related to change. In the microservice paradigm, a system is structured by composing small independent building blocks, each with a dedicated persistence tool and communicating exclusively via message passing. In this kind of organization, the complexity is moved to the level of coordination of services (often called orchestration [9]).

Two questions are often asked: *"are Microservices just tiny services?"* and *"Are Microservices yet another name for SOA?"*. A Microservice is not just a *tiny service*. Each microservice is expected to implement a single *business capability*, in fact a very limited system functionality, bringing benefits in terms of service maintainability and extendability. Since each microservice represents a single business capability, which is delivered and updated independently, discovering bugs or adding minor improvements do not have any impact on other services and on their releases. In common practice, it is also expected that a single service can be developed and managed by a single team [10]. The idea to have a team working on a single microservice is rather appealing: to build a system with a modular and loosely coupled design, one should pay attention to the organization structure and its communication patterns as they, according to Conway's Law [3], directly impact the produced design. So if one creates an organization with each team working on a single service, such structure will make the communication more efficient not only on the team level, but within the whole organization, improving the resulting design in terms of modularity.

Microservices is not yet another name for SOA. Indeed, there are some notable differences. In SOA, services are not required to be self-contained with data and User Interface, and their own persistence tools, eg. database. SOA has no focus on independent deployment units and related consequences, it is simply an approach for business-to-business intercommunication. The idea of SOA was to enable business-level programming through business processing engines and languages such as WS-BPEL and BPMN that were built on top of the vast literature on business modelling [11]. Furthermore, the emphasis was all on *service orchestration* more than service development and deployment.

In this paper, we report the experience of migration from monolithic to microservice of the Danske Banks *FX Core* system. The documentation of the original system architecture was sparse and the vast majority of technical details have been obtained by direct conversations, interviews and discussions with the *FX Core* team, and by manually inspecting the source code. This was a lengthy process given the complexity of the

original monolithic architecture, never thoroughly documented before, and here we can report some aspects of this analysis. However, for the information to be publicly available, all the confidential details such as concrete names of protocols, external providers and specific services has been withheld. Furthermore, the internal logic of certain components could not be described in depth.

II. Danske Bank *FX Core* system

Foreign Exchange, often abbreviated as *forex* or *FX*, is the exchange of currencies, i.e. the conversion from one currency to another. Exchange of currencies is of interest to both private individuals, corporations, financial institutions and governments. *FX* encompasses everything from private transactions performed in foreign countries (e.g. Internet shopping from abroad and use of credit cards while traveling) to corporations moving their financial assets from one currency to another and exporting or importing products to and from foreign markets. *FX* has grown with globalization and it is now globally the largest financial market in the world, averaging a daily transaction volume of roughly 5 trillion dollars. This results in some transactions reaching the hundred millions of dollars. Unlike the stock exchange, there is no centralized market, instead *FX* is decentralized and done *over-the-counter (OTC)*, i.e. traders negotiate prices and trade directly between each other. Traders are typically the largest multinational banks, trading on behalf of their customers or themselves. Additionally, due to the decentralized and global nature of *FX*, the market is open 24 hours a day, five days a week [7].

The *FX IT* (Fig 1) system is part of the banks *Corporates and Institutions (C&I)* department and handles price streaming, trades, line-checks and associated tasks, such as analytics and post-trade management. *FX IT* acts as a gateway between the international markets and Danske Banks clients, including their own traders. C&I's clients are mainly large financial institutions and large multi-national corporations. They continuously process streams of currency pair prices from the markets on which they calculate margins to reduce risk, especially important on *swaps* and *forwards*, before streaming final prices to clients. Clients can then act on a price by registering a trade or check if they have the required collateral with line-checks.

The *FX Core* system is part of *FX IT* and it handles trades and line-checks. This includes registration, validation and post-trade management. Below there is a brief description of the two main responsibilities of *FX Core*.

LineChecks are used to check whether a client has the financial collateral to perform a trade and how a trade will affect said collateral, also called their *Line*. This collateral can be a multitude of financial assets, e.g. stocks, bonds or cash. Line-checks are always executed as part of a trade, but is also run separately, so Danske Banks traders can ensure that their customers are capable of requested trades.

Trades are received from both Danske Banks clients and *external providers*, i.e. external clients and markets. The trade is then validated and line-checked, before being registered. Depending on the type of trade, the trade is either done immediately, i.e. a *spot trade*, or registered in the system as a contract for future execution, i.e. *swaps* and *forwards*. When the trade is executed it involves moving the financial assets between banking books, i.e. from one account to another. After a trade has been registered a number of actions can be executed on it, e.g. multiple trades can be joined to ease administration or be split into smaller trades to reduce margins, *forward* and *swap* contracts can be extended or pre-settled and trades can be corrected or deleted by internal clients. Additionally, the system can also run batch jobs in order to balance books between departments or to analyze trades, to e.g. detect fraudulent behavior such as money laundering.

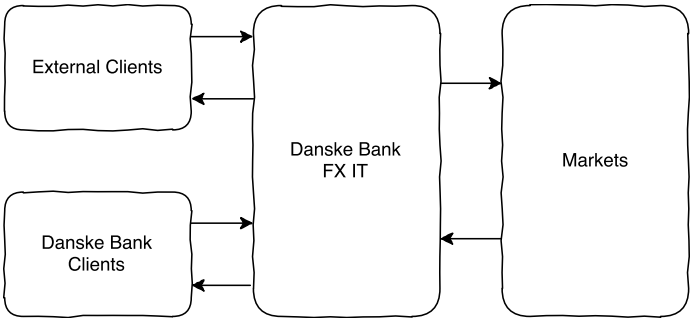

Fig. 1. FX IT handles both price streaming and requests for trades and line-checks from global markets, e.g. other banks, pension funds and large corporations. Prices of currency pairs are streamed to FX IT, which then calculates prices of specific trades, before streaming them to external and internal clients. The clients can then request FX IT for trades or line-checks on the prices they have received. These clients are usually used by Danske Banks internal traders and external customers. Additionally, trade and line-check requests can also be received from the markets, when banks wish to exchange currencies directly. *FX Core* is part of *FX IT*, but handles tasks associated with trades and line-checks, thus not handling any of the price streaming and stream processing.

III. FX Core Monolith

Danske Banks monolithic system is presented in Figure 2. The services are deployable individually and are replicated and deployed across a cluster. The system also utilizes APIs as interfaces for clients to interact with the services of the system, and a messaging system to delegate received requests from external providers. Despite of these solutions to facilitate scalability, Danske Bank has experienced severe challenges when trying to rapidly develop the system and deploying consistent changes, and in general in handling system complexity. Full details on the architecture can be found in [4].

The monolithic architecture is componentized in a variety of ways. The system utilizes services, shared software libraries and thick desktop clients and it is deployed on three Windows Server hosts, located at the three Danske Banks data center locations. Each of the system's components can be deployed individually, as they are independent processes, but in fact

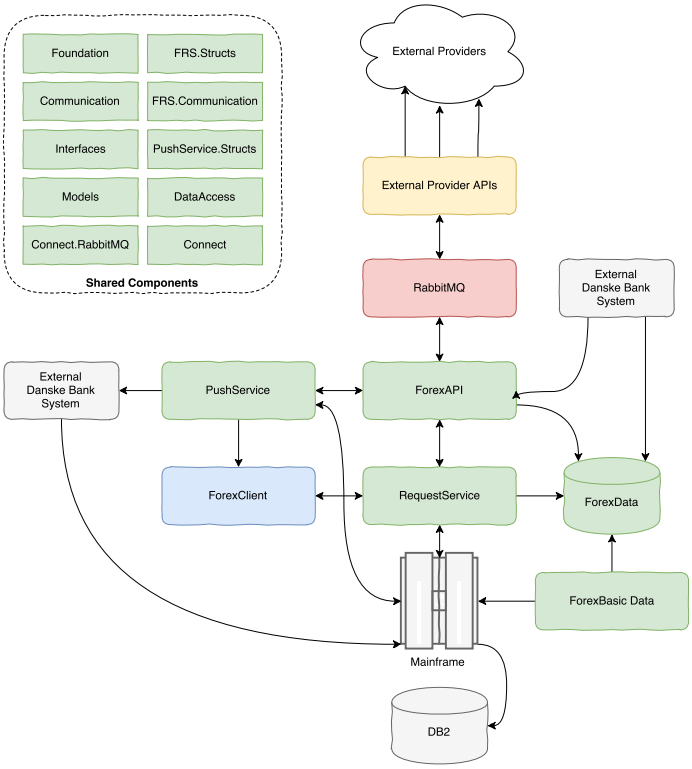

Fig. 2. Danske Banks monolithic architecture. Red services are infrastructure services, green are part of the monolith, blue is the client, yellow are external provider APIs and grey are external Danske Bank systems.The components of the system integrate directly with each other, resulting in many different communication technologies and high coupling. The external provider APIs are part of the monolith and consist of multiple services, with each one connecting to a different provider. Their names have been excluded due to confidentiality. The shared components are used across almost all services and are also internally dependent on each other. The ForexData database is one big monolithic MS SQL database, shared amongst many of the monolithic components and also accessed by external systems

they are always co-located as a whole system for availability reasons and for the components to be highly coupled.

IV. FX Core Microservice Architecture

The Danske Banks new *FXCore* architecture is based on the microservice architectural style and is is intended to completely replace the old monolithic architecture. The overall infrastructure is depicted in Figure 3.

Danske Banks *FX Core* microservice architecture is hosted on private data-centers, i.e. not in the cloud. This means that new hosts can not be provisioned and de-provisioned as rapidly and automated as in a cloud. It is in their interest to provide a private cloud for systems to run in, but due to regulations on banking data, this is still work in progress. There are three data-center locations in Denmark, which can be utilized to achieve better availability and increased resilience to the internal systems.

On the IT departments roadmap is the adoption of the *Red Hat OpenShift* [6] Iaas/PaaS platform, on the internal data-centers. However, at the moment, the infrastructure consists

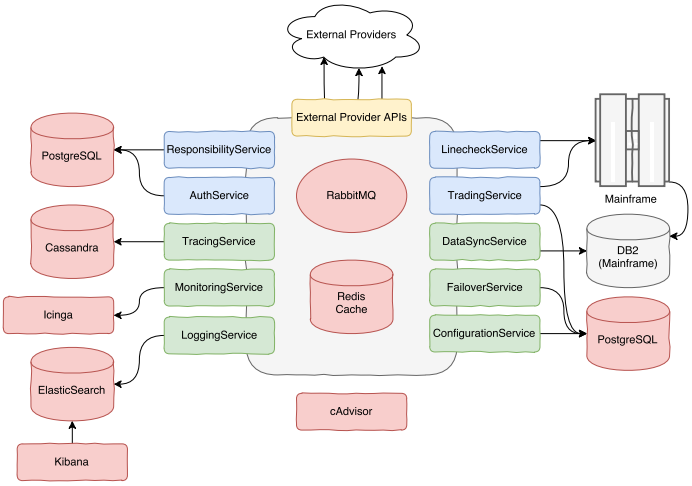

Fig. 3. The new *FX Core* microservice architecture. Red services are infrastructure services, green are foundation services, blue are business services and the yellow is external provider APIs. All non-infrastructure services communicate via messaging over *RabbitMQ* and have direct access to the *Redis cache*, which is used to cache data from *DB2*. Databases in the diagram should be seen as database management systems (DBMS), meaning that although four services use *PostgreSQL* they all have their own standalone database within the DBMS

of VM's ordered through a web-portal, and which are setup manually by the *FX Core* team.

V. Conclusions

The re-engineering of the system discussed in this paper led to *reduced complexity*, *lower coupling*, *higher cohesion* and a *simplified integration*. The large components of the monolithic architecture, which were highly coupled, had overlapping responsibilities and integrated in a multitude of ways, have been substituted by several independent microservices. As a direct consequence, the size of the services is now generally smaller when compared to the large components of the monolith. Since services implement focused functionalities, even their names reveal to a large extent their responsibilities, detail which was previously absent. Services have now *reduced feature overlapping*. For example, functionalities such as trade-registration and line-checks in the monolithic architecture were handled by both *ForexAPI* and *RequestService*. The reason of this had to be found in the historical development of the system. After the analysis and the re-engineering, the microservice architecture features a *TradingService* and a *LineCheckService* that independently handle these functions. Furthermore, with a *polyglot architecture*, i.e. not technology dependent, the development team is no longer dependent on the *.NET* platform or MS SQL databases. Instead, services can be implemented in any language.

The future will see a growing attention regarding the matters discussed here and the development of new programming languages intended to address the microservice paradigm [5]. Languages for microservices should be able to model microservices in a *uniform way* and at a level of abstraction that also allows for their easy interconnection [2].

Microservice composition techniques are needed and have to be used when: (a) frequent revision of microservices are needed, (b) changes in existing offered functionalities (i.e. microservices behavior), and adjustment of business policies and objectives (i.e., composition requirements) are required [1]. Microservices-based systems must be self-adaptive according to the available microservices in the specific execution context and to the changes affecting its execution. To guarantee self-specialization (i.e., automatic selection and composition of microservices), languages for microservices must include the adaptation aspect as an inner characteristich of the system design (i.e, Adaptive by-Design [2]) and not an exception to manage.

REFERENCES

[1] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik. A context-aware framework for dynamic composition of process fragments in the internet of services. *J. Internet Services and Applications*, 8(1):6:1–6:23, 2017.

[2] A. Bucchiarone, M. De Sanctis, A. Marconi, M. Pistore, and P. Traverso. Incremental composition for adaptive by-design service based systems. In *IEEE ICWS 2016*, pages 236–243, 2016.

[3] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[4] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. Microservices: Migration of a mission critical system. https://arxiv.org/abs/1704.04173.

[5] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi. Microservices: a language-based approach. In *Present and Ulterior Software Engineering*. Springer, 2017.

[6] Red Hat. Openshift: Paas by red hat, built on docker and kubernetes. https://www.openshift.com/.

[7] A. MacEachern. How are international exchange rates set? http://www.investopedia.com/ask/answers/forex/how-forex-exchange-rates-set.asp.

[8] Matthew C. MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, Rebekah Metz, and Booz Allen Hamilton. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.

[9] M. Mazzara and S. Govoni. *A Case Study of Web Services Orchestration*, pages 1–16. Springer Berlin Heidelberg, 2005.

[10] N.Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In Bertrand Meyer and Manuel Mazzara, editors, *Present and Ulterior Software Engineering*. Springer, 2017.

[11] Z. Yan, M. Mazzara, E. Cimpian, and A. Urbanec. Business process modeling: Classifications and perspectives. In *BPSC 2007.*, page 222, 2007.